Object Semantics

6.170 Lecture 2

Spring 2004

The objectives of this lecture are:

- to help you become familiar with the basic runtime mechanism common to all object-oriented languages (but with a particular focus on Java): variables, object references, assignments, mutability, and so on;
- to introduce a diagrammatic notation, *object diagrams*, for describing particular configurations, or *snapshots*, of objects in the heap;[1]
- to make you aware, in passing, of some tricky issues that we'll return to later in more detail, and which turn out to be of fundamental importance: equality, rep invariants, and rep exposure.

When you've completed this material, you should have a solid grasp of what happens when Java code executes, so that you can predict what some code will do without running it.

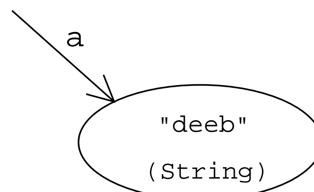
1 Variables, References and Objects

Some types of objects can be created with *literals*. What happens when you run this?

```
1.      String a = "deeb";
2.      System.out.println (a);
```

It prints `deeb`. Statement 1 is a declaration (of the variable `a`) and an assignment (to the variable `a`) all in one. The expression `"deeb"` is called a string literal. Statement 2 is a procedure call that prints a representation of `a` to the standard output stream; don't worry about its details for now.

We can draw the result of the first statement as an *object diagram*, showing that `a` is a *reference* to an object of type `String`.



Note that `a` is a reference to the string; it is not the string itself, nor is it a slot that holds the string. This is an important point to understand, though it makes no difference in this small example.

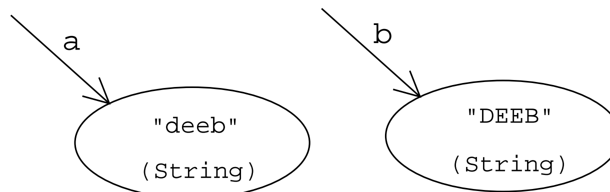
[1]Later, we will introduce *object models* that are far more useful and which describe sets of heap snapshots.

What happens when you run this?

```
1.      String a = "deeb";
2.      String b = a.toUpperCase ();
3.      System.out.println (b);
```

It prints `DEEB`. Statement 2 is a call to the method `toUpperCase`. The method has a *receiver* `a`. The call results in the creation of a fresh string that is then bound to the variable `b`.

We can draw the result of the first two statements as an *object diagram*, showing that `a` is a *reference* to an object of type `String`; similarly for `b`.



2 Immutability and Mutability

What does this do?

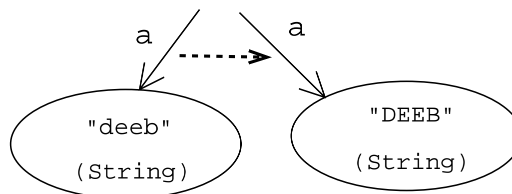
```
1.      String a = "deeb";
2.      a.toUpperCase ();
3.      System.out.println (a);
```

It prints `deeb`. Statement 2 creates a fresh string that gets thrown away since it is bound to no variable. The string object referenced by `a` is not changed: strings are *immutable*.

What about this?

```
1.      String a = "deeb";
2.      a = a.toUpperCase ();
3.      System.out.println (a);
```

Again, no object changes. But `a` is made to refer to the new object by the assignment in Statement 2, so the result is `DEEB`.



Now let's look at another class in the Java standard library. The `Date` class represents a date (and a time, too, but for simplicity we'll ignore that).

What does this code do?

```
1.      Date now = new Date ();
2.      System.out.println (now);
```

Statement 1 declares the variable `now` and assigns to it a Date object. Since Java doesn't have date literals, however, we must create the Date object using a different beast: a *constructor*. A constructor is a

kind of procedure whose name is always the name of the class (e.g. `Date`), and which may take arguments to initialize the new object (here, no arguments are needed). A constructor is called using the `new` keyword.

In this case, the `Date` constructor with no arguments creates a Date object initialized to today's date. So Statement 2 would print today's date, February 4, 2004.

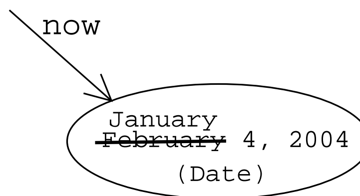
What if we added two more statements:

```
3.      now.setMonth (0);
4.      System.out.println (now);
```

Unlike the `toUpperCase` method we used on `Strings`, the `setMonth` method in Statement 3 doesn't return a new object. Instead, it changes, or *mutates*, its receiver object. `Dates`, unlike `Strings`, can change after they have been created, and thus are said to be *mutable*.

The parameter of `setMonth` is an integer in the range $0 \ldots 11$, where 0 means January. So the effect of this code is to print January 4, 2004.

Here's how the object diagram for this example would look, after Statement 3:



3 Aliasing and Reference Equality

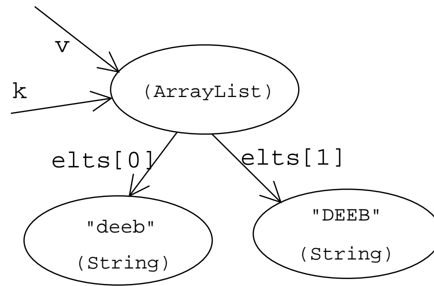
Java provides a variety of collections as part of its standard library. An `ArrayList` is like an array — a sequence of objects — but it can grow and shrink dynamically. Here's an example that uses `ArrayList`:

```
1.      ArrayList v = new ArrayList ();
2.      ArrayList k = v;
3.      String a = "deeb";
4.      v.add (a);
5.      k.add (a.toUpperCase());
6.      System.out.println (v.lastElement ());
```

The method `lastElement` is like the string method `toUpperCase`: it has no effect on the receiver and returns a reference to an object (in this case, the last element of the `ArrayList v`). The method `add`, on the other hand, takes an argument, the string `a`, and adds it to the end of the list.

The code above prints `DEEB`, since after Statement 2 the two variables `k` and `v` are names for the same `ArrayList` object: they are said to be *aliases*. The calls to `add` mutate the single `ArrayList` object, first adding the object for the lower case string, then the upper case string. The changes are visible through both names, so in Statements 4 to 6 we could actually permute the names `k` and `v` without any change in behavior.

Here is the object diagram:



Aliasing is pervasive in languages like Java, and very useful. But it adds a lot of complexity. For one thing, it breaks the rule that a statement 'affects only the variables it mentions'. Just because `v` isn't mentioned in Statement 5 doesn't mean that it won't affect the result of Statement 6 which mentions only `v` and not `k`.

How can we observe the aliasing more directly? By testing equality:

```
ArrayList v = new ArrayList ();
ArrayList k = v;
if (v == k)
    System.out.println ("same");
else
    System.out.println ("different");
```

which results in `same` being printed. The built-in `==` test tells you whether two references are for the same object, so it's often called a test of *reference equality*.

```
ArrayList v = new ArrayList ();
ArrayList k = new ArrayList ();
if (v == k)
    System.out.println ("same");
else
    System.out.println ("different");
```

This code prints `different`, because `v` and `k` are distinct objects. It's a fundamental property of constructors that the objects they return really are fresh. In fact, the garbage collector can recycle an object, but only if there is no reference to it still around. This ensures that even if objects are recycled, we can never tell.

What if we use strings:

```
String a = "deeb";
String b = a.toUpperCase ();
String c = "DEEB";
if (b == c)
    System.out.println ("same");
else
    System.out.println ("different");
```

This prints `different`, because `toUpperCase` returns one fresh object, and the string literal `"DEEB"` returns a different one. Even though the content of the strings look the same (`DEEB`), referential equality distinguishes between them.

Now for a puzzle: what does this do?

```
String a = "DEEB";
String b = "DEEB";
if (a == b)
    System.out.println ("same");
else
    System.out.println ("different");
```

Strangely, this prints `same`, because the Java virtual machine automatically *interns* string literals. If it can tell that two string literals have the same sequence of characters, it only allocates one object. You'd be right to think this is a bit confusing; it's a performance optimization.

In fact, it's very bad form to test reference equality of immutable objects, unless you're doing something subtle with memory management. Arguably it's a design defect of Java that you can even observe whether two immutable objects are the same or not. So how should you compare two immutable objects? With an `equals` method.

The `String` class provides a method `equals` that tells you whether two strings contain the same sequence of characters or not. This code

```
String a = "deeb";
String b = a.toUpperCase ();
if (b.equals ("DEEB"))
    System.out.println ("same characters");
else
    System.out.println ("different characters");
```

prints `same characters`.

When we study inheritance, you'll learn that every class automatically inherits an `equals` method, so you might think you don't need to write one. But it's almost never what you want, so whenever you design a class, one of the first things you'll need to figure out is when two objects of the class should be considered equal to one another.

Here are some questions:

- Would you expect that generally `x == y` implies `x.equals (y)`? Yes, it should. Because the `equals` method can be user-defined, just like any other method, you could make it behave in any way you wanted. On a mutable type, it might even mutate the object! But that would be disastrous: there's a generic contract that clients expect `equals` to obey. More on this later.

- Why would a language have immutable types? Because aliasing is complicated, and when you use immutable types, the issue doesn't arise. Also, code built with immutable types can sometimes be more efficient.

- So if immutable types are so much simpler, why have mutable types? Because mutation gives a very useful form of modularity: it allows you to make local changes to a structure. And mutation is often a natural way to model entities in the real world: a transaction on a bank account changes it; it doesn't produce a new bank account.

4 Null References

What does this do?

```
String a = null;
System.out.println (a);
```

It prints `null`. The keyword `null` denotes a value that can be taken on by an object reference. It means that the reference does not in fact refer to any object. There is no null object!

But note that this code

```
1.        String a = null;
2.        String b = a.toUpperCase ();
3.        System.out.println (b);
```

behaves quite differently. It throws a `NullPointerException` on Statement 2. We'll learn about exceptions later, but for now, all you need to understand is that Statement 2 failed, when the expression `a.toUpperCase()` was evaluated.

What's the difference? The receiver to a method call can never be null, because it identifies the object that receives the call—and that has to be some object. So `a.toUpperCase()` fails when `a` is null. But in the previous example, `System.out.println(a)` is OK when `a` is null, since the null reference is an argument, not the receiver. (What is the receiver of `System.out.println`?)

You can write a method that tests whether an argument is null and does something appropriate. Dereferencing null is a common programming mistake in Java. To avoid it, you can check whether a reference is null before you attempt to call a method.

In general, rather than catching nulls and treating them specially, it's better to avoid creating null references in the first place. You'll learn about that when we discuss representation invariants. Sometimes you can't avoid it, and then it's important to document where the null references may occur. That's one reason specifications are important: they can spare you runtime errors and unnecessary checks.

Here are some questions:

- In general, would you expect `a.equals (b)` to be substitutable for `b.equals (a)` ? No, because when `a` is null and `b` is not, the first will throw an exception, and the second will (usually) return false.

- OK, smarty pants, leave nulls alone. Would you then expect `a.equals (b)` and `b.equals (a)` to have the same effect? Yes, you would. In fact, this property of the `equals` method — called *symmetry* — is demanded by Java's *object contract*, the specification that well-designed objects must satisfy. We'll see later when we study equality in depth what other properties are required, and how easy it is to mess up and write an `equals` method that does *not* have these properties.

5 User-defined Classes and Fields

So far we've only used objects built into Java. Now let's make some objects of our own:

```
class Trans {
      int amount;
      Date date;
}
```

This code declares a *class*, a kind of template for making objects. These objects are going to represent transactions in a banking system. Each object has an integer amount (which may be negative for a withdrawal), and a date/time stamp to mark the moment at which the transaction occurred. The class declares

two *fields* or *instance variables*, amount and date. Each object of the class will contain two references, one to an integer and one to a date. The type Date is a class from the Java library; it's predefined like String (but not part of the language definition the way String is).

The type int is a rather strange beast. It's not a class at all, but a *primitive type*. Variables or fields of type int don't hold references to integer objects; they hold the integers themselves. You may think it a bit jarring that an object-oriented language has this rather unobject-oriented notion in it (and many people share your opinion). Sometimes we'll actually need an integer that's an object, and in that case we can use the class Integer from the Java library. How do you get from an int to an Integer and back? To create an Integer, you use a constructor:

```
int_i = 5;
Integer obj_i = new Integer (i);
```

and to extract the primitive integer from an integer object, you call a method:

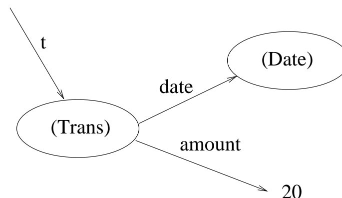
```
int i = obj_i.intValue();
```

A little cumbersome, so that's one reason people don't like this design.

If we run this code:

```
1.      Trans t = new Trans ();
2.      t.amount = 20;
3.      t.date = new Date ();
```

a fresh object gets created, and its fields are set, resulting in this configuration:



The expression on the right-hand side of Statement 1 is a call to a constructor: it makes a new object with default values for the fields. In this case, it creates a Trans object with zero for the amount and null for date. Statement 2 is called a *setter*: it sets the value of the field amount of the object referred to by t. Statement 3 has another constructor call on the right; creating a new Date, which by default creates a Date object representing the moment at which the object is itself created. But it's also a setter: it sets the date field of t to point to this new date.

6 User-defined Constructors

So we've succeeded in making a Trans object representing a deposit of twenty dollars at this moment in time. The way we did it – creating an uninitialized object and then setting its fields – is not a good one, however. We'll want our transactions to be well-formed; for example we won't want to have transactions that don't have dates. And perhaps we'll want every transaction to have a non-zero amount. Later, we'll study these kinds of *invariants* in much more depth.

For now, just observe that immediately after Statement 1 we have a transaction object that is *not* well formed. When an object is created, its fields are initialized to default values: null for object references, and zero for integers. So t.amount will be zero, and t.date will be null. These default values are rarely what you want; after all, which values make sense will depend on the problem we're trying to solve. In this case, you'd have to know something about banking to know that a transaction of zero dollars is ill-formed.

Is it a big deal that there's a bogus transaction object hanging around between Statements 1 and 2? Yes, it is, and here's why. We'd like the responsibility for ensuring that objects of the class Trans are well formed to be handled entirely within the Trans class. In our program, you need to check not only the code of the class, but also the code that uses the class. At this scale, it's not a disaster. But in a much larger program, you need as much *modularity* as you can get, confining tricky aspects of the program as much as possible to small areas of the code.

To solve this problem, we declare our own constructor:

```
class Trans {
       int amount;
       Date date;
       Trans (int a, Date d) {amount = a; date = d;}
}
```

This constructor takes an amount and a date as arguments, and creates a transaction object with that amount and date. If I'd wanted to ensure that the amount of a transaction is non-zero, I could have added a check that threw an exception if the amount was zero; we'll see how to do that later.

A peculiar, but useful, property of constructors is that having defined our own constructor, the default constructor – the one that just initializes each field to default values – becomes no longer available. So Statement 1 will no longer compile. Instead, we can write

```
Trans t = new Trans (20, new Date ());
```

and that one line of code will have the effect that Statements 1 to 3 had previously.

(We haven't fully solved the problem of modularizing the invariant of Trans, by the way. You can still mess up a Trans object by setting its date field to null from outside, for example. The first step to prevent this is to make use of Java's accessibility mechanisms: we can make the fields *private* so they can't be read or written from outside the class. In fact, this turns out not to be enough, as we'll learn when we study data abstraction.)

7 Conclusion

Now we've seen all the key notions for how objects are manipulated in an object-oriented language. We've seen how they're created; how references are bound to objects; and how their fields are set. We've mentioned the two fundamental kinds of equality – reference equality (tested with ==) and object equality (tested with an equals method – about which we'll have much more to say later. These mechanisms comprise one important part of what it means to be *object oriented*.